Displaying Bitmaps Efficiently

Learn how to use common techniques to process and load[Bitmap](file:///G:\Studio\SDK\docs\reference\android\graphics\Bitmap.html) objects in a way that keeps your user interface (UI) components responsive and avoids exceeding your application memory limit. If you're not careful, bitmaps can quickly consume your available memory budget leading to an application crash due to the dreaded exception:  
java.lang.OutofMemoryError: bitmap size exceeds VM budget.

There are a number of reasons why loading bitmaps in your Android application is tricky:

* Mobile devices typically have constrained system resources. Android devices can have as little as 16MB of memory available to a single application. The[Android Compatibility Definition Document](http://source.android.com/compatibility/downloads.html) (CDD),*Section 3.7. Virtual Machine Compatibility* gives the required minimum application memory for various screen sizes and densities. Applications should be optimized to perform under this minimum memory limit. However, keep in mind many devices are configured with higher limits.
* Bitmaps take up a lot of memory, especially for rich images like photographs. For example, the camera on the [Galaxy Nexus](http://www.android.com/devices/detail/galaxy-nexus) takes photos up to 2592x1936 pixels (5 megapixels). If the bitmap configuration used is [ARGB\_8888](file:///G:\Studio\SDK\docs\reference\android\graphics\Bitmap.Config.html) (the default from the Android 2.3 onward) then loading this image into memory takes about 19MB of memory (2592\*1936\*4 bytes), immediately exhausting the per-app limit on some devices.
* Android app UI’s frequently require several bitmaps to be loaded at once. Components such as[ListView](file:///G:\Studio\SDK\docs\reference\android\widget\ListView.html), [GridView](file:///G:\Studio\SDK\docs\reference\android\widget\GridView.html) and [ViewPager](file:///G:\Studio\SDK\docs\reference\android\support\v4\view\ViewPager.html) commonly include multiple bitmaps on-screen at once with many more potentially off-screen ready to show at the flick of a finger.

# Loading Large Bitmaps Efficiently

Images come in all shapes and sizes. In many cases they are larger than required for a typical application user interface (UI). For example, the system Gallery application displays photos taken using your Android devices's camera which are typically much higher resolution than the screen density of your device.

Given that you are working with limited memory, ideally you only want to load a lower resolution（分辨率） version in memory. The lower resolution version should match the size of the UI component that displays it. An image with a higher resolution does not provide any visible benefit, but still takes up precious memory and incurs additional performance overhead due to additional on the fly scaling.

This lesson walks you through decoding large bitmaps without exceeding the per application memory limit by loading a smaller subsampled version in memory.

## Read Bitmap Dimensions and Type

The [BitmapFactory](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.html) class provides several decoding methods ([decodeByteArray()](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.html#decodeByteArray(byte[], int, int, android.graphics.BitmapFactory.Options)), [decodeFile()](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.html#decodeFile(java.lang.String, android.graphics.BitmapFactory.Options)),[decodeResource()](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.html#decodeResource(android.content.res.Resources, int, android.graphics.BitmapFactory.Options)), etc.) for creating a [Bitmap](file:///G:\Studio\SDK\docs\reference\android\graphics\Bitmap.html) from various sources. Choose the most appropriate decode method based on your image data source. These methods attempt to allocate memory for the constructed bitmap and therefore can easily result in an OutOfMemory exception. Each type of decode method has additional signatures that let you specify decoding options via the[BitmapFactory.Options](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.Options.html) class. Setting the [inJustDecodeBounds](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.Options.html#inJustDecodeBounds) property to true while decoding avoids memory allocation, returning null for the bitmap object but setting [outWidth](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.Options.html#outWidth), [outHeight](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.Options.html#outHeight)and [outMimeType](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.Options.html#outMimeType). This technique allows you to read the dimensions and type of the image data prior to construction (and memory allocation) of the bitmap.

BitmapFactory.Options options = new BitmapFactory.Options();  
options.inJustDecodeBounds = true;  
BitmapFactory.decodeResource(getResources(), R.id.myimage, options);  
int imageHeight = options.outHeight;  
int imageWidth = options.outWidth;  
String imageType = options.outMimeType;

To avoid java.lang.OutOfMemory exceptions, check the dimensions of a bitmap before decoding it, unless you absolutely trust the source to provide you with predictably sized image data that comfortably fits within the available memory.

## Load a Scaled Down Version into Memory

Now that the image dimensions are known, they can be used to decide if the full image should be loaded into memory or if a subsampled version should be loaded instead. Here are some factors to consider:

* Estimated memory usage of loading the full image in memory.
* Amount of memory you are willing to commit to loading this image given any other memory requirements of your application.
* Dimensions of the target [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html) or UI component that the image is to be loaded into.
* Screen size and density of the current device.

For example, it’s not worth loading a 1024x768 pixel image into memory if it will eventually be displayed in a 128x96 pixel thumbnail in an [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html).

To tell the decoder to subsample the image, loading a smaller version into memory, set[inSampleSize](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.Options.html#inSampleSize) to true in your [BitmapFactory.Options](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.Options.html) object. For example, an image with resolution 2048x1536 that is decoded with an [inSampleSize](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.Options.html#inSampleSize) of 4 produces a bitmap of approximately 512x384. Loading this into memory uses 0.75MB rather than 12MB for the full image (assuming a bitmap configuration of [ARGB\_8888](file:///G:\Studio\SDK\docs\reference\android\graphics\Bitmap.Config.html)). Here’s a method to calculate a sample size value that is a power of two based on a target width and height:

public static int calculateInSampleSize(  
            BitmapFactory.Options options, int reqWidth, int reqHeight) {  
    // Raw height and width of image  
    final int height = options.outHeight;  
    final int width = options.outWidth;  
    int inSampleSize = 1;  
  
    if (height > reqHeight || width > reqWidth) {  
  
        final int halfHeight = height / 2;  
        final int halfWidth = width / 2;  
  
        // Calculate the largest inSampleSize value that is a power of 2 and keeps both  
        // height and width larger than the requested height and width.  
        while ((halfHeight / inSampleSize) > reqHeight  
                && (halfWidth / inSampleSize) > reqWidth) {  
            inSampleSize \*= 2;  
        }  
    }  
  
    return inSampleSize;  
}

**Note:** A power of two value is calculated because the decoder uses a final value by rounding down to the nearest power of two, as per the [inSampleSize](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.Options.html#inSampleSize) documentation.

To use this method, first decode with [inJustDecodeBounds](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.Options.html#inJustDecodeBounds) set to true, pass the options through and then decode again using the new [inSampleSize](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.Options.html#inSampleSize) value and [inJustDecodeBounds](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.Options.html#inJustDecodeBounds) set to false:

public static Bitmap decodeSampledBitmapFromResource(Resources res, int resId,  
        int reqWidth, int reqHeight) {  
  
    // First decode with inJustDecodeBounds=true to check dimensions  
    final BitmapFactory.Options options = new BitmapFactory.Options();  
    options.inJustDecodeBounds = true;  
    BitmapFactory.decodeResource(res, resId, options);  
  
    // Calculate inSampleSize  
    options.inSampleSize = calculateInSampleSize(options, reqWidth, reqHeight);  
  
    // Decode bitmap with inSampleSize set  
    options.inJustDecodeBounds = false;  
    return BitmapFactory.decodeResource(res, resId, options);  
}

This method makes it easy to load a bitmap of arbitrarily large size into an [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html) that displays a 100x100 pixel thumbnail, as shown in the following example code:

mImageView.setImageBitmap(  
    decodeSampledBitmapFromResource(getResources(), R.id.myimage, 100, 100));

You can follow a similar process to decode bitmaps from other sources, by substituting the appropriate [BitmapFactory.decode\*](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.html#decodeByteArray(byte[], int, int, android.graphics.BitmapFactory.Options)) method as needed.

# Processing Bitmaps Off the UI Thread

The [BitmapFactory.decode\*](file:///G:\Studio\SDK\docs\reference\android\graphics\BitmapFactory.html#decodeByteArray(byte[], int, int, android.graphics.BitmapFactory.Options)) methods, discussed in the [Load Large Bitmaps Efficiently](file:///G:\Studio\SDK\docs\training\displaying-bitmaps\load-bitmap.html) lesson, should not be executed on the main UI thread if the source data is read from disk or a network location (or really any source other than memory). The time this data takes to load is unpredictable and depends on a variety of factors (speed of reading from disk or network, size of image, power of CPU, etc.). If one of these tasks blocks the UI thread, the system flags your application as non-responsive and the user has the option of closing it (see [Designing for Responsiveness](file:///G:\Studio\SDK\docs\guide\practices\responsiveness.html) for more information).

This lesson walks you through processing bitmaps in a background thread using [AsyncTask](file:///G:\Studio\SDK\docs\reference\android\os\AsyncTask.html) and shows you how to handle concurrency issues.

## Use an AsyncTask

The [AsyncTask](file:///G:\Studio\SDK\docs\reference\android\os\AsyncTask.html) class provides an easy way to execute some work in a background thread and publish the results back on the UI thread. To use it, create a subclass and override the provided methods. Here’s an example of loading a large image into an [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html) using [AsyncTask](file:///G:\Studio\SDK\docs\reference\android\os\AsyncTask.html) and[decodeSampledBitmapFromResource()](file:///G:\Studio\SDK\docs\training\displaying-bitmaps\load-bitmap.html#decodeSampledBitmapFromResource):

class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {  
    private final WeakReference<ImageView> imageViewReference;  
    private int data = 0;  
  
    public BitmapWorkerTask(ImageView imageView) {  
        // Use a WeakReference to ensure the ImageView can be garbage collected  
        imageViewReference = new WeakReference<ImageView>(imageView);  
    }  
  
    // Decode image in background.  
    @Override  
    protected Bitmap doInBackground(Integer... params) {  
        data = params[0];  
        return decodeSampledBitmapFromResource(getResources(), data, 100, 100));  
    }  
  
    // Once complete, see if ImageView is still around and set bitmap.  
    @Override  
    protected void onPostExecute(Bitmap bitmap) {  
        if (imageViewReference != null && bitmap != null) {  
            final ImageView imageView = imageViewReference.get();  
            if (imageView != null) {  
                imageView.setImageBitmap(bitmap);  
            }  
        }  
    }  
}

The [WeakReference](file:///G:\Studio\SDK\docs\reference\java\lang\ref\WeakReference.html) to the [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html) ensures that the [AsyncTask](file:///G:\Studio\SDK\docs\reference\android\os\AsyncTask.html) does not prevent the [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html)and anything it references from being garbage collected. There’s no guarantee the [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html) is still around when the task finishes, so you must also check the reference in [onPostExecute()](file:///G:\Studio\SDK\docs\reference\android\os\AsyncTask.html#onPostExecute(Result)). The[ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html) may no longer exist, if for example, the user navigates away from the activity or if a configuration change happens before the task finishes.

To start loading the bitmap asynchronously, simply create a new task and execute it:

public void loadBitmap(int resId, ImageView imageView) {  
    BitmapWorkerTask task = new BitmapWorkerTask(imageView);  
    task.execute(resId);  
}

## Handle Concurrency

Common view components such as [ListView](file:///G:\Studio\SDK\docs\reference\android\widget\ListView.html) and [GridView](file:///G:\Studio\SDK\docs\reference\android\widget\GridView.html) introduce another issue when used in conjunction with the [AsyncTask](file:///G:\Studio\SDK\docs\reference\android\os\AsyncTask.html) as demonstrated in the previous section. In order to be efficient with memory, these components recycle child views as the user scrolls. If each child view triggers an [AsyncTask](file:///G:\Studio\SDK\docs\reference\android\os\AsyncTask.html), there is no guarantee that when it completes, the associated view has not already been recycled for use in another child view. Furthermore, there is no guarantee that the order in which asynchronous tasks are started is the order that they complete.

The blog post [Multithreading for Performance](http://android-developers.blogspot.com/2010/07/multithreading-for-performance.html) further discusses dealing with concurrency, and offers a solution where the [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html) stores a reference to the most recent [AsyncTask](file:///G:\Studio\SDK\docs\reference\android\os\AsyncTask.html) which can later be checked when the task completes. Using a similar method, the [AsyncTask](file:///G:\Studio\SDK\docs\reference\android\os\AsyncTask.html) from the previous section can be extended to follow a similar pattern.

Create a dedicated [Drawable](file:///G:\Studio\SDK\docs\reference\android\graphics\drawable\Drawable.html) subclass to store a reference back to the worker task. In this case, a[BitmapDrawable](file:///G:\Studio\SDK\docs\reference\android\graphics\drawable\BitmapDrawable.html) is used so that a placeholder image can be displayed in the [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html) while the task completes:

static class AsyncDrawable extends BitmapDrawable {  
    private final WeakReference<BitmapWorkerTask> bitmapWorkerTaskReference;  
  
    public AsyncDrawable(Resources res, Bitmap bitmap,  
            BitmapWorkerTask bitmapWorkerTask) {  
        super(res, bitmap);  
        bitmapWorkerTaskReference =  
            new WeakReference<BitmapWorkerTask>(bitmapWorkerTask);  
    }  
  
    public BitmapWorkerTask getBitmapWorkerTask() {  
        return bitmapWorkerTaskReference.get();  
    }  
}

Before executing the [BitmapWorkerTask](file:///G:\Studio\SDK\docs\training\displaying-bitmaps\process-bitmap.html#BitmapWorkerTask), you create an [AsyncDrawable](file:///G:\Studio\SDK\docs\training\displaying-bitmaps\process-bitmap.html#AsyncDrawable) and bind it to the target[ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html):

public void loadBitmap(int resId, ImageView imageView) {  
    if (cancelPotentialWork(resId, imageView)) {  
        final BitmapWorkerTask task = new BitmapWorkerTask(imageView);  
        final AsyncDrawable asyncDrawable =  
                new AsyncDrawable(getResources(), mPlaceHolderBitmap, task);  
        imageView.setImageDrawable(asyncDrawable);  
        task.execute(resId);  
    }  
}

The cancelPotentialWork method referenced in the code sample above checks if another running task is already associated with the [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html). If so, it attempts to cancel the previous task by calling [cancel()](file:///G:\Studio\SDK\docs\reference\android\os\AsyncTask.html#cancel(boolean)). In a small number of cases, the new task data matches the existing task and nothing further needs to happen. Here is the implementation of cancelPotentialWork:

public static boolean cancelPotentialWork(int data, ImageView imageView) {  
    final BitmapWorkerTask bitmapWorkerTask = getBitmapWorkerTask(imageView);  
  
    if (bitmapWorkerTask != null) {  
        final int bitmapData = bitmapWorkerTask.data;  
        // If bitmapData is not yet set or it differs from the new data  
        if (bitmapData == 0 || bitmapData != data) {  
            // Cancel previous task  
            bitmapWorkerTask.cancel(true);  
        } else {  
            // The same work is already in progress  
            return false;  
        }  
    }  
    // No task associated with the ImageView, or an existing task was cancelled  
    return true;  
}

A helper method, getBitmapWorkerTask(), is used above to retrieve the task associated with a particular [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html):

private static BitmapWorkerTask getBitmapWorkerTask(ImageView imageView) {  
   if (imageView != null) {  
       final Drawable drawable = imageView.getDrawable();  
       if (drawable instanceof AsyncDrawable) {  
           final AsyncDrawable asyncDrawable = (AsyncDrawable) drawable;  
           return asyncDrawable.getBitmapWorkerTask();  
       }  
    }  
    return null;  
}

The last step is updating onPostExecute() in [BitmapWorkerTask](file:///G:\Studio\SDK\docs\training\displaying-bitmaps\process-bitmap.html#BitmapWorkerTask) so that it checks if the task is cancelled and if the current task matches the one associated with the [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html):

class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {  
    ...  
  
    @Override  
    protected void onPostExecute(Bitmap bitmap) {  
        **if (isCancelled()) {  
            bitmap = null;  
        }**  
  
        if (imageViewReference != null && bitmap != null) {  
            final ImageView imageView = imageViewReference.get();  
            **final BitmapWorkerTask bitmapWorkerTask =  
                    getBitmapWorkerTask(imageView);**  
            if (**this == bitmapWorkerTask &&** imageView != null) {  
                imageView.setImageBitmap(bitmap);  
            }  
        }  
    }  
}

This implementation is now suitable for use in [ListView](file:///G:\Studio\SDK\docs\reference\android\widget\ListView.html) and [GridView](file:///G:\Studio\SDK\docs\reference\android\widget\GridView.html) components as well as any other components that recycle their child views. Simply call loadBitmap where you normally set an image to your [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html). For example, in a [GridView](file:///G:\Studio\SDK\docs\reference\android\widget\GridView.html) implementation this would be in the[getView()](file:///G:\Studio\SDK\docs\reference\android\widget\Adapter.html#getView(int, android.view.View, android.view.ViewGroup)) method of the backing adapter.

# Caching Bitmaps

Loading a single bitmap into your user interface (UI) is straightforward, however things get more complicated if you need to load a larger set of images at once. In many cases (such as with components like [ListView](file:///G:\Studio\SDK\docs\reference\android\widget\ListView.html), [GridView](file:///G:\Studio\SDK\docs\reference\android\widget\GridView.html) or[ViewPager](file:///G:\Studio\SDK\docs\reference\android\support\v4\view\ViewPager.html)), the total number of images on-screen combined with images that might soon scroll onto the screen are essentially unlimited.

Memory usage is kept down with components like this by recycling the child views as they move off-screen. The garbage collector also frees up your loaded bitmaps, assuming you don't keep any long lived references. This is all good and well, but in order to keep a fluid and fast-loading UI you want to avoid continually processing these images each time they come back on-screen. **A memory and disk cache can often help here**, allowing components to quickly reload processed images.

This lesson walks you through using a memory and disk bitmap cache to improve the responsiveness and fluidity of your UI when loading multiple bitmaps.

## Use a Memory Cache

A memory cache offers fast access to bitmaps at the cost of taking up valuable application memory. The [LruCache](file:///G:\Studio\SDK\docs\reference\android\util\LruCache.html) class (also available in the [Support Library](file:///G:\Studio\SDK\docs\reference\android\support\v4\util\LruCache.html) for use back to API Level 4) is particularly well suited to the task of caching bitmaps, keeping recently referenced objects in a strong referenced [LinkedHashMap](file:///G:\Studio\SDK\docs\reference\java\util\LinkedHashMap.html) and evicting the least recently used member before the cache exceeds its designated size.

**Note:** In the past, a popular memory cache implementation was a [SoftReference](file:///G:\Studio\SDK\docs\reference\java\lang\ref\SoftReference.html) or [WeakReference](file:///G:\Studio\SDK\docs\reference\java\lang\ref\WeakReference.html)bitmap cache, however this is not recommended. Starting from Android 2.3 (API Level 9) the garbage collector is more aggressive with collecting soft/weak references which makes them fairly ineffective. In addition, prior to Android 3.0 (API Level 11), the backing data of a bitmap was stored in native memory which is not released in a predictable manner, potentially causing an application to briefly exceed its memory limits and crash.

In order to choose a suitable size for a [LruCache](file:///G:\Studio\SDK\docs\reference\android\util\LruCache.html), a number of factors should be taken into consideration, for example:

* How memory intensive is the rest of your activity and/or application?
* How many images will be on-screen at once? How many need to be available ready to come on-screen?
* What is the screen size and density of the device? An extra high density screen (xhdpi) device like [Galaxy Nexus](http://www.android.com/devices/detail/galaxy-nexus) will need a larger cache to hold the same number of images in memory compared to a device like [Nexus S](http://www.android.com/devices/detail/nexus-s) (hdpi).
* What dimensions and configuration are the bitmaps and therefore how much memory will each take up?
* How frequently will the images be accessed? Will some be accessed more frequently than others? If so, perhaps you may want to keep certain items always in memory or even have multiple [LruCache](file:///G:\Studio\SDK\docs\reference\android\util\LruCache.html)objects for different groups of bitmaps.
* Can you balance quality against quantity? Sometimes it can be more useful to store a larger number of lower quality bitmaps, potentially loading a higher quality version in another background task.

There is no specific size or formula that suits all applications, it's up to you to analyze your usage and come up with a suitable solution. A cache that is too small causes additional overhead with no benefit, a cache that is too large can once again cause java.lang.OutOfMemory exceptions and leave the rest of your app little memory to work with.

Here’s an example of setting up a [LruCache](file:///G:\Studio\SDK\docs\reference\android\util\LruCache.html) for bitmaps:

private LruCache<String, Bitmap> mMemoryCache;  
  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    ...  
    // Get max available VM memory, exceeding this amount will throw an  
    // OutOfMemory exception. Stored in kilobytes as LruCache takes an  
    // int in its constructor.  
    final int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024);  
  
    // Use 1/8th of the available memory for this memory cache.  
    final int cacheSize = maxMemory / 8;  
  
    mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {  
        @Override  
        protected int sizeOf(String key, Bitmap bitmap) {  
            // The cache size will be measured in kilobytes rather than  
            // number of items.  
            return bitmap.getByteCount() / 1024;  
        }  
    };  
    ...  
}  
  
public void addBitmapToMemoryCache(String key, Bitmap bitmap) {  
    if (getBitmapFromMemCache(key) == null) {  
        mMemoryCache.put(key, bitmap);  
    }  
}  
  
public Bitmap getBitmapFromMemCache(String key) {  
    return mMemoryCache.get(key);  
}

**Note:** In this example, one eighth of the application memory is allocated for our cache. On a normal/hdpi device this is a minimum of around 4MB (32/8). A full screen [GridView](file:///G:\Studio\SDK\docs\reference\android\widget\GridView.html) filled with images on a device with 800x480 resolution would use around 1.5MB (800\*480\*4 bytes), so this would cache a minimum of around 2.5 pages of images in memory.

When loading a bitmap into an [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html), the [LruCache](file:///G:\Studio\SDK\docs\reference\android\util\LruCache.html) is checked first. If an entry is found, it is used immediately to update the [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html), otherwise a background thread is spawned to process the image:

public void loadBitmap(int resId, ImageView imageView) {  
    final String imageKey = String.valueOf(resId);  
  
    final Bitmap bitmap = getBitmapFromMemCache(imageKey);  
    if (bitmap != null) {  
        mImageView.setImageBitmap(bitmap);  
    } else {  
        mImageView.setImageResource(R.drawable.image\_placeholder);  
        BitmapWorkerTask task = new BitmapWorkerTask(mImageView);  
        task.execute(resId);  
    }  
}

The [BitmapWorkerTask](file:///G:\Studio\SDK\docs\training\displaying-bitmaps\process-bitmap.html#BitmapWorkerTask) also needs to be updated to add entries to the memory cache:

class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {  
    ...  
    // Decode image in background.  
    @Override  
    protected Bitmap doInBackground(Integer... params) {  
        final Bitmap bitmap = decodeSampledBitmapFromResource(  
                getResources(), params[0], 100, 100));  
        addBitmapToMemoryCache(String.valueOf(params[0]), bitmap);  
        return bitmap;  
    }  
    ...  
}

## Use a Disk Cache

A memory cache is useful in speeding up access to recently viewed bitmaps, however you cannot rely on images being available in this cache. Components like [GridView](file:///G:\Studio\SDK\docs\reference\android\widget\GridView.html) with larger datasets can easily fill up a memory cache. Your application could be interrupted by another task like a phone call, and while in the background it might be killed and the memory cache destroyed. Once the user resumes, your application has to process each image again.

A disk cache can be used in these cases to persist processed **bitmaps and help decrease loading times where images are no longer available in a memory cache**. Of course, fetching images from disk is slower than loading from memory and should be done in a background thread, as disk read times can be unpredictable.

**Note:** A [ContentProvider](file:///G:\Studio\SDK\docs\reference\android\content\ContentProvider.html) might be a more appropriate place to store cached images if they are accessed more frequently, for example in an image gallery application.

The sample code of this class uses a DiskLruCache implementation that is pulled from the [Android source](https://android.googlesource.com/platform/libcore/+/jb-mr2-release/luni/src/main/java/libcore/io/DiskLruCache.java). Here’s updated example code that adds a disk cache in addition to the existing memory cache:

private DiskLruCache mDiskLruCache;  
private final Object mDiskCacheLock = new Object();  
private boolean mDiskCacheStarting = true;  
private static final int DISK\_CACHE\_SIZE = 1024 \* 1024 \* 10; // 10MB  
  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    ...  
    // Initialize memory cache  
    ...  
    // Initialize disk cache on background thread  
    **File cacheDir = getDiskCacheDir(this, DISK\_CACHE\_SUBDIR);**  
    new InitDiskCacheTask().execute(cacheDir);  
    ...  
}  
  
class InitDiskCacheTask extends AsyncTask<File, Void, Void> {  
    @Override  
    protected Void doInBackground(File... params) {  
        synchronized (mDiskCacheLock) {  
            File cacheDir = params[0];  
            mDiskLruCache = DiskLruCache.open(cacheDir, DISK\_CACHE\_SIZE);  
            mDiskCacheStarting = false; // Finished initialization  
            mDiskCacheLock.notifyAll(); // Wake any waiting threads  
        }  
        return null;  
    }  
}  
  
class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {  
    ...  
    // Decode image in background.  
    @Override  
    protected Bitmap doInBackground(Integer... params) {  
        final String imageKey = String.valueOf(params[0]);  
  
        // Check disk cache in background thread  
        Bitmap bitmap = getBitmapFromDiskCache(imageKey);  
  
        if (bitmap == null) { // Not found in disk cache  
            // Process as normal  
            final Bitmap bitmap = decodeSampledBitmapFromResource(  
                    getResources(), params[0], 100, 100));  
        }  
  
        // Add final bitmap to caches  
        addBitmapToCache(imageKey, bitmap);  
  
        return bitmap;  
    }  
    ...  
}  
  
public void addBitmapToCache(String key, Bitmap bitmap) {  
    // Add to memory cache as before  
    if (getBitmapFromMemCache(key) == null) {  
        mMemoryCache.put(key, bitmap);  
    }  
  
    // Also add to disk cache  
    synchronized (mDiskCacheLock) {  
        if (mDiskLruCache != null && mDiskLruCache.get(key) == null) {  
            mDiskLruCache.put(key, bitmap);  
        }  
    }  
}  
  
public Bitmap getBitmapFromDiskCache(String key) {  
    synchronized (mDiskCacheLock) {  
        // Wait while disk cache is started from background thread  
        while (mDiskCacheStarting) {  
            try {  
                mDiskCacheLock.wait();  
            } catch (InterruptedException e) {}  
        }  
        if (mDiskLruCache != null) {  
            return mDiskLruCache.get(key);  
        }  
    }  
    return null;  
}  
  
// Creates a unique subdirectory of the designated app cache directory. Tries to use external  
// but if not mounted, falls back on internal storage.  
public static File getDiskCacheDir(Context context, String uniqueName) {  
    // Check if media is mounted or storage is built-in, if so, try and use external cache dir  
    // otherwise use internal cache dir  
    final String cachePath =  
            Environment.MEDIA\_MOUNTED.equals(Environment.getExternalStorageState()) ||  
                    !isExternalStorageRemovable() ? getExternalCacheDir(context).getPath() :  
                            context.getCacheDir().getPath();  
  
    return new File(cachePath + File.separator + uniqueName);  
}

**Note:** Even initializing the disk cache requires disk operations and therefore should not take place on the main thread. However, this does mean there's a chance the cache is accessed before initialization. To address this, in the above implementation, a lock object ensures that the app does not read from the disk cache until the cache has been initialized.

While the memory cache is checked in the UI thread, the disk cache is checked in the background thread. Disk operations should never take place on the UI thread. When image processing is complete, the final bitmap is added to both the memory and disk cache for future use.

## Handle Configuration Changes

Runtime configuration changes, such as a screen orientation change, cause Android to destroy and restart the running activity with the new configuration (For more information about this behavior, see [Handling Runtime Changes](file:///G:\Studio\SDK\docs\guide\topics\resources\runtime-changes.html)). You want to avoid having to process all your images again so the user has a smooth and fast experience when a configuration change occurs.

Luckily, you have a nice memory cache of bitmaps that you built in the [Use a Memory Cache](file:///G:\Studio\SDK\docs\training\displaying-bitmaps\cache-bitmap.html#memory-cache) section. This cache can be passed through to the new activity instance using a [Fragment](file:///G:\Studio\SDK\docs\reference\android\app\Fragment.html) which is preserved by calling [setRetainInstance(true)](file:///G:\Studio\SDK\docs\reference\android\app\Fragment.html#setRetainInstance(boolean))). After the activity has been recreated, this retained[Fragment](file:///G:\Studio\SDK\docs\reference\android\app\Fragment.html) is reattached and you gain access to the existing cache object, allowing images to be quickly fetched and re-populated into the [ImageView](file:///G:\Studio\SDK\docs\reference\android\widget\ImageView.html) objects.

Here’s an example of retaining a [LruCache](file:///G:\Studio\SDK\docs\reference\android\util\LruCache.html) object across configuration changes using a [Fragment](file:///G:\Studio\SDK\docs\reference\android\app\Fragment.html):

private LruCache<String, Bitmap> mMemoryCache;  
  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    ...  
    RetainFragment retainFragment =  
            RetainFragment.findOrCreateRetainFragment(getFragmentManager());  
    mMemoryCache = retainFragment.mRetainedCache;  
    if (mMemoryCache == null) {  
        mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {  
            ... // Initialize cache here as usual  
        }  
        retainFragment.mRetainedCache = mMemoryCache;  
    }  
    ...  
}  
  
class RetainFragment extends Fragment {  
    private static final String TAG = "RetainFragment";  
    public LruCache<String, Bitmap> mRetainedCache;  
  
    public RetainFragment() {}  
  
    public static RetainFragment findOrCreateRetainFragment(FragmentManager fm) {  
        RetainFragment fragment = (RetainFragment) fm.findFragmentByTag(TAG);  
        if (fragment == null) {  
            fragment = new RetainFragment();  
            fm.beginTransaction().add(fragment, TAG).commit();  
        }  
        return fragment;  
    }  
  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        **setRetainInstance(true);**  
    }  
}

To test this out, try rotating a device both with and without retaining the [Fragment](file:///G:\Studio\SDK\docs\reference\android\app\Fragment.html). You should notice little to no lag as the images populate the activity almost instantly from memory when you retain the cache. Any images not found in the memory cache are hopefully available in the disk cache, if not, they are processed as usual.